Ada 9X Project Report

The Fixed-Point Facility in Ada

February 1990

Office of the Under Secretary of Defense for Acquisition

Washington, D.C. 20301

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 1990	Final Report

4. TITLE AND SUBTITLE

Ada 9X Project Report, The Fixed-Point Facility in Ada, February 1990

5. FUNDING NUMBERS

C = MDA-903-87

6. AUTHOR(S)

Robert B.K. Dewar
John B. Goodenough, editor

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER

SEI-90-SR-2

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Ada Joint Program Office
1211 South Fern St., 3E113
The Pentagon
Washington, DC 20301-3080

Ada 9X Project Office
AF Armament Lab/FXG
Eglin AFB, Florida 32542-5434

10. SPONSORING/MONITORING REPORT NUMBER

11. SUPPLEMENTARY NOTES

This report has been produced under the sponsorship of the Ada 9X Project Office. It is one in a series that addresses special issues relevant to the Ada revision effort.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report addresses a number of issues involving fixed-point arithmetic in Ada. A number of possible Ada 9X requirements are suggested for consideration.

Ada includes a facility for declaration of an arithmetic on fixed-point values. The model is deliberately intended to be as close as possible to the Ada floating-point model, including the notion of model numbers and model intervals. The major difference is that the error is absolute, rather than relative; in other words, the model numbers are spaced evenly in the fixed-point case.

14. SUBJECT TERMS

Ada 9X, fixed-point, floating-point, Ada Joint Program Office, Ada 9X Project Office

15. NUMBER OF PAGES

42

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF
UNCLASSIFIED	UNCLASSIFIED	UL	

Ada 9X Project Report

The Fixed-Point Facility in Ada

February 1990

Office of the Under Secretary of Defense for Acquisition

Washington, D.C. 20301

SEI-90-SR-2
February 1990

The Fixed-Point Facility in Ada

Robert B. K. Dewar

New York University

This report has been produced under the sponsorship of the Ada 9X Project Office. It is one in a series that addresses special issues relevant to the Ada revision effort. John B. Goodenough, of the Software Engineering Institute, has served as the editor and coordinator for each report.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This work is sponsored by the U.S. Department of Defense. The views and conclusions contained in this document are solely those of the author(s) and should not be interpreted as representing official policies, either expressed or implied, of Carnegie Mellon University, the U.S. Air Force, the Department of Defense, or the U.S. Government.

Table of Contents

1. The Fixed-Point Facility in Ada

Abstract: *This report addresses a number of issues involving fixed-point arithmetic in Ada. A number of possible Ada 9X requirements are suggested for consideration.*

Ada includes a facility for declaration of and arithmetic on fixed-point values. The model is deliberately intended to be as close as possible to the Ada floating-point model, including the notion of model numbers and model intervals. The major difference is that the error is absolute, rather than relative; in other words, the model numbers are spaced evenly in the fixed-point case.

This fixed-point facility is intended for a variety of uses:

1. "Poor man's floating-point." On machines where floating-point hardware is either non-existent, or very inefficient, the use of Ada floating-point may be impractical. Ada requires that floating-point arithmetic be present on all Ada implementations, but if this is achieved by software simulation, there will be cases where the use of floating-point is impractical for efficiency reasons.

   On such machines, the use of fixed-point provides a capability for carrying out calculations with fractional quantities more efficiently. Even when reasonably efficient hardware floating-point is available, fixed-point arithmetic is often faster, and in some real-time applications which are computation-bound at critical points, this difference may be significant.

2. Increased accuracy. In a situation where the absolute error control of fixed-point is acceptable, e.g., a case where the range of values to be dealt with is limited, fixed-point provides more accuracy than floating point for a given word length, since no space is wasted for the exponent.

3. Mapping data from specialized hardware and other external data. There are a number of situations in which data from external devices is naturally in fixed-point format, for instance a volt-meter may return a voltage in units of 2^{-8} volts. In such cases it is both efficient and convenient to deal with the data directly in fixed-point format.

4. A special, but important, case of external world fixed-point data arises in fiscal calculations, where quantities of money are typically decimal scaled fixed-point values (e.g., $56.34). In commercial programs which deal with money, it is much more convenient to deal with such quantities directly in fixed-point form. The use of floating-point here is not practical because the decimal values encountered are not floating-point model numbers, and hence unexpected rounding and truncation errors can occur (e.g., 0.10 * 10 is not necessarily equal to 1.00)

2. The Representation of Fixed-Point Numbers

The intention behind the fixed-point design in Ada is that fixed-point values be represented internally as integers. The value is interpreted as the product of the integer value stored and the *small* value. The *small* value is always known at compile time and is associated with the fixed-point type. This means that it does not need to be stored with each instance of a fixed-point value.

The *Reference Manual for the Ada Programming Language* (RM) does not enforce this method of representing fixed-point numbers since it generally is not in the business of specifying internal representations. Even in the case of integers, the RM does not specify the exact method of internal representation. However, it is likely that this method of storing fixed-point numbers will in effect be mandated by an interpretation of the chapter 13 facilities, which would permit declarations like:

```
type FIXED_1 is delta 0.25 range -1.00 .. +0.75;
for FIXED_1'SMALL use 0.25;
for FIXED_1'SIZE  use 3;
```

If implementations are expected to accept these representation clauses, then clearly objects of type FIXED_1 can only be stored as integer multiples of *small*, as shown by the following table:

Value	Corresponding Integer	Representation in Binary
-1.00	-4	100
-0.75	-3	101
-0.50	-2	110
-0.25	-1	111
0.00	0	000
+0.25	+1	001
+0.50	+2	010
+0.75	+3	011

Here we are assuming that integers are represented in twos-complement, which is of course a correct assumption for the great majority of machines. On a ones-complement machine, the binary representations would be different, and in particular, the actual represented range would be -0.75..+0.75. However, it seems likely that the intention is that the correspondence between the fixed-point values and the corresponding integers *would* hold in all implementations, so that for example, UNCHECKED_CONVERSION between entries in the first two columns would work "as expected."

It should be repeated that there is nothing in the RM that requires this correspondence, and individual implementations may choose, at least in some circumstances, to use different representation approaches. In particular, the use of hardware decimal formats may be convenient in some situations, and this is discussed later on (see page 11). However, the underlying expectation that all fixed-point values are represented as integers is fundamental and is a model that we shall generally refer to in this discussion.

2.1. The Meaning of the *small* Value

There are four quantities that are relevant in the representation of fixed-point quantities:

1. The declared delta, i.e., the value of delta used in the declaration of the fixed-point type.
2. The actual delta, which is the largest power of 2 that does not exceed the declared delta.
3. The value of *small*, which is the distance between model numbers. In the absence of a representation clause specifying *small*, it may be, but need not be, equal to the actual delta.
4. The value of SAFE_SMALL, the distance between safe numbers, which are the model numbers of the base type. The safe numbers constrain the accuracy of fixed-point calculations. Following the lead of the floating-point model, the RM permits fixed-point values to be represented with more accuracy than that required by the safe numbers, which means that the unit value of a fixed-point representation can be smaller than SAFE_SMALL.

The consequence of the fourth point is that whether or not *small* (and therefore, the value of SAFE_SMALL) is set by a representation clause, the implementation might choose to represent fixed-point values with extra accuracy. However, the ARG is considering an AI (AI-00341) which removes this flexibility, in part, by requiring that a specified *small* value be the one which is actually used as the unit position in the represented values. The AI does not, however, address whether representations with different accuracies are allowed for the same type when *small* is *not* specified explicitly. This issue is discussed in Section 2.2.

POSSIBLE ADA 9X REQUIREMENT

An explicit specification of small for a fixed-point type should specify the machine precision used to hold all values of the type. No intermediate results can be held with increased accuracy.

Discussion: This creates a (deliberate) inconsistency between floating-point and fixed-point, which means the wording dealing with both together must be carefully be examined. This inconsistency is justified, because in the case of floating-point, but not fixed-point, the hardware formats may effectively dictate the use of extra precision.

Compatibility considerations: None. This is upwards compatible with the original RM, and consistent with the proposed ARG interpretation.

In the remainder of this report, we will assume that a specified value of *small* for a fixed-point type determines the accuracy of represented values of the type.

2.2. The Selection of SAFE_SMALL

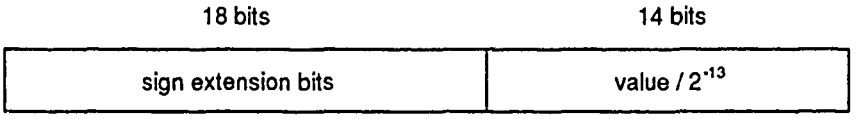
Even with the agreement that SAFE_SMALL in effect specifies the accuracy of fixed-point representations, there is still a quite deliberate freedom left to implementations to choose a SAFE_SMALL that is less than *small*. Consider the following example:

```
type FIXED_2 is delta 2.0**(-13) range -1.0 .. +1.0;
```

The actual delta is 2^{-13} and this is also the value of FIXED_2'SMALL, in the absence of a length clause specifying *small* (RM 3.5.9(6, 10)). The value of FIXED_2'BASE'SMALL (which is the same as FIXED_2'SAFE_SMALL) cannot be greater than this actual value but it *can* be smaller. Assuming that we have a 32-bit machine, it may be convenient and efficient (in the absence of a SIZE clause) for FIXED_2 objects to be represented using 32 bits, so that ordinary 32-bit arithmetic can be used for operations on values of this type.

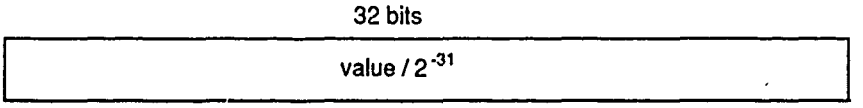
The FIXED_2 type only requires 14 bits, so the question is how to use the remaining 18 bits in the word. There are two basic approaches:

1. Keep SAFE_SMALL equal to FIXED_2'SMALL, leaving unused bits (typically these would be sign-extension bits) in the high order positions:



   For example, the fixed-point value 2^{-13} is represented as the integer 1. This approach is similar to what would typically be used for integer values with a range less than the available hardware range.

2. Make SAFE_SMALL as small as possible, using the extra bits to provide extra precision. In this example, SAFE_SMALL can be equal to 2^{-31}.



   The fixed-point value 2^{-13} is represented as the integer $2^{-13} / 2^{-31} = 2^{18}$. This approach has the advantage of providing extra precision without impacting the efficiency of calculations using the fixed-point type.

Of course these two approaches are only extremes of a range of possible implementations which provide intermediate precisions. However, there seems to be no good reason to choose intermediate cases, so these are ignored here. There are good arguments in favor of both approaches, which we can summarize as follows:

In favor of approach 1

Allowing, but not requiring the implementation to provide extra precision introduces unnecessary implementation differences, with the usual negative impact on portability. There are many situations in which the extra precision is actively unhelpful, and would

have to be suppressed by the use of explicit *small* clauses.[1] If a program needs the extra precision, then it should use *small* clauses to force this extra precision. This can be done in a machine independent manner by appropriate use of attributes.[2]

The appeal to floating-point as a model for the behavior of fixed-point is inappropriate, since the reason we allow the extra precision in the floating-point case, even though it introduces portability problems, is that these portability problems are fundamental, given the different hardware implementations of floating-point. Furthermore, typical users of floating-point understand that the precision is implementation dependent. It is much less obvious that this expectation holds in the fixed-point case.

In favor of approach 2

The extra precision provided by this approach is free in terms of the efficiency of arithmetic operations. It seems clearly beneficial to give more accurate results where possible, as is routinely done in the floating-point case. In those situations where extra accuracy is harmful, or where absolute portability of results is required, SMALL clauses can be used to control exactly what is required.

The use of attributes to declare these "extra precision" representations is possible, but very clumsy, and it seems much preferable to give the extra precision as the default behavior, requiring the relatively straightforward SMALL declarations only in the relatively unusual case where they are required. Note in particular that many of the cases where the extra precision is unwelcome (e.g., in fiscal calculations) it is necessary anyway to specify the *small* because it is not a power of 2.

Providing the extra precision may actually increase the efficiency of the generated code by simplifying overflow checks. In the case where the range of the fixed point definition covers a natural power of 2 range, then by left justifying the value, overflow in the fixed point calculations corresponds to overflow in the underlying integer arithmetic, which can be usually be detected much more cheaply than the explicit range check which is needed if the value is right justified.

Both these arguments have merit. However, if we favor the second argument, then it would probably be better to mandate the extra precision. One problem not addressed here (but treated later; see page 13) is that the extra precision approach potentially makes the value of SAFE_SMALL dependent on the SIZE of the type, which causes some further anomalies.

---

[1]For example, suppose you are performing fiscal calculations to the nearest cent. If values are reported to customers to the nearest penny but are carried in the computer with greater accuracy, then sums will not add up correctly. For example, if an account with a reported balance of $1.03 is combined with accounts reported to have balances of $1.15 and $1.02, you would expect the combined total to be $3.20. But if the values are carried with more accuracy, e.g., $1.033 and $1.153 and $1.024, the reported total will be $3.210. In this case, it is unhelpful to carry the balances with increased accuracy. So the programmer needs to be able to control whether extra accuracy is used.

[2]For example, suppose we wish to ensure that FIXED_2 values are represented with maximum precision. We know that FIXED_2 values are represented using 32 bits since FIXED_2'SIZE is 32. The minimum number of bits required to represent FIXED_2 model numbers is 14, which is the value of FIXED_2'MANTISSA + 1 (since 'MANTISSA does not include the sign bit). Since 'SIZE includes the sign bit and 'MANTISSA doesn't, the number of extra bits available for increased precision is (FIXED_2'SIZE - (FIXED_2'MANTISSA + 1), i.e., 17 bits. To use these extra bits for increased accuracy, we need to specify that the value of FIXED_2'SMALL is divided by $2^{\text{extra bits}}$. So we can declare a related type:

```
type ACCURATE_FIXED_2 is delta 2.0**(-13) range -1.0 .. +1.0;
for ACCURATE_FIXED_2'SMALL use FIXED_2'SMALL / 2.0 ** (FIXED_2'SIZE - FIXED_2'MANTISSA - 1);
```

Which argument appeals largely depends on the use to which fixed-point is being put. If it is serving as "poor-man's floating-point" (as is usually the case in embedded applications), then the extra precision is clearly welcome. If on the other hand, fixed-point is being used to get precisely scaled integer results (as for financial applications), the extra precision is unwelcome. In any case there seems to be unnecessary implementation freedom.

POSSIBLE ADA 9X REQUIREMENT

Ada 9X should be more specific in directing implementations how to choose fixed point base types, with the aim of improving portability of fixed-point code.

Discussion: The arguments for and against the provision of extra precision should be evaluated and a decision made as to which default behavior is appropriate. It is possible to consider the addition of a pragma or other upwards compatible declarative method to allow the programmer to specify the required approach. From a language point of view, such a pragma is probably overkill, since the mechanism of specifying small explicitly is provided. However, compatibility considerations, as discussed below, may make the provision of such a feature pragmatically desirable.

Compatibility considerations: Technically none, since either behavior is permitted by the RM. However, in practice, there may be programs that depend on one or the other approach. This is technically improper, but not unlikely. Such programs might be negatively impacted by a requirement that the implementation they are using be changed. Programs can avoid such impact by explicitly specifying small for all fixed-point types.

2.3. Reduced Accuracy Subtypes

Consider the set of declarations:

```
type FIXED_3 is delta 2.0**(-8) range -128.0 .. +128.0;
subtype FIXED_4 is FIXED_3 delta 2.0**(-7);
```

The current Ada definition certainly permits this set of declarations, and the model numbers of the FIXED_4 subtype are a subset of the model numbers of FIXED_3 even though FIXED_3 and FIXED_4 have the same value of SAFE_SMALL. Suppose that FIXED_3's base type has been chosen to give no extra accuracy, i.e., FIXED_3'SAFE_SMALL equals FIXED_3'SMALL equals 2^{-8}. Can values of subtype FIXED_4 be stored with model number accuracy, i.e., with one less bit of accuracy? A similar issue has been discussed at length for floating-point (see AI-407).

The disadvantage of allowing reduced accuracy representations of this kind is that it means an assignment such as:

```
F4 : FIXED_3;
F5 : FIXED_4;

F5 := F4;
```

can cause loss of accuracy, which is not a normally expected consequence of an assignment. This loss of accuracy can occur in an even less apparent manner when a generic is instantiated with the reduced accuracy subtype.

Reduced accuracy representations can also possibly occur in records if record component clauses are allowed to eliminate bits that are "unnecessary" for representing model numbers:

```
type REC is
    record
        PRECISE  : FIXED_3;
        REDUCED : FIXED_4;
    end record;

for REC use
    record
        PRECISE at 0 range 0..15;
        REDUCED at 2 range 1..15;          -- legal?
    end record;
```

FIXED_4'MANTISSA is 14, meaning that 15 bits are sufficient to hold all model numbers of the subtype. The ARG has not yet ruled on the status of such a record component clause; it is unclear whether the RM allows such a range in a component clause, since it changes the accuracy with which values of the base type are stored. Of course, if the above component clause is accepted by the implementation, and X is an object of type REC, then accuracy can be lost in this assignment:

```
X.REDUCED := X.PRECISE;
```

The component clause for REDUCED would be illegal if *small* for FIXED_3 were explicitly specified to be 2^{-8}, given our assumption that a specified value for *small* determines the accuracy of all represented values (see Section 2.1).

Now suppose SAFE_SMALL for FIXED_3 were 2^{-24} instead of 2^{-8}, i.e., suppose FIXED_3's base type occupies 32 bits and the extra bits are used to provide extra precision. Could the above record representation clause still be accepted? Note that in this case, the component PRECISE would have a reduced accuracy as compared to stored values of type FIXED_3 outside of the record type. Accepting the component clause would imply a reduced accuracy representation even for a first named subtype.

It is not at all clear that reduced accuracy representations provide an important functionality, and the question of whether or not they are permitted should be addressed.

POSSIBLE ADA 9X REQUIREMENT

Ada 9X should clarify the status of reduced accuracy representations for fixed-point types. In particular, Ada 9X should consider whether to require that SAFE_SMALL for a fixed-point type determines the machine precision used for all values of the type (both stored and computed).

Discussion: Generally it seems like an unnecessary source of non-portable behavior and a subtle source of program error to allow implementations to provide reduced accuracy fixed-point representations.

Compatibility considerations: Technically none. Even if Ada 9X were to forbid the implementation of reduced accuracy representations, no problems would arise, since implementations are currently free to ignore such a request, so any correct Ada program cannot depend on this feature.

2.4. The Values of *small* That Must Be Supported

There is a continued confusion in the current language over the issue of what values of *small* must be supported by an implementation. The ACVC suite has been extremely cautious in this area. (ACVC 1.11 contains a single very simple test with a decimal *small* value, but under pressure from implementors, this test has been reclassified as a DEP (dependent) test, meaning that implementations may reject even this simple case.)

There are three general categories of *small* values which arise:

- Powers of 2. Support for these is clearly required, and all compilers implement powers of 2. However, the range of powers of 2 which are supported varies, and is quite restrictive in some compilers.
- Powers of 10. The need for these cases arises in conjunction with fiscal calculations, where the arithmetic must be done using accurate decimal scaling.
- Other values. These are less common. They typically arise in conjunction with external devices which provide scaled data values.

The current RM seems to indicate that all possible *small* values should be supported. The only possible reason for not supporting particular *small* values would have to be based on hardware considerations, and the issue of implementing fixed-point is essentially hardware independent, since the underlying operations required are simply integer arithmetic, which is presumably available on all machines.

However, compilers have been slow to implement even the simple non-binary cases. The impact, particularly on the ability to use Ada for fiscal calculations is negative. Implementing the simple cases of decimal *smalls* is straightforward. Even the multiplication and division case, as long as only decimal *small* values are involved (even if they are not all the same), is relatively simple (similar operations are required in all COBOL compilers).

POSSIBLE ADA 9X REQUIREMENT

The range of required small values should be clearly specified. Ada 9X should support the requirements of fiscal calculations with regard to the implementation of decimal small values.

Discussion: At least decimal small values should be required to meet the requirements of fiscal calculations. The issue of whether general small values should be required is more contentious. A situation in which some compilers support certain small values and others do not creates a major portability problem. If an Ada compiler fails to support the small values specified by an Ada program, the job of adapting the program to the new compiler may be extremely difficult and involve major modifications. It seems desirable to enforce a greater level of uniformity.

Compatibility considerations: Probably none, unless Ada 9X goes so far as to prohibit the acceptance of small values now accepted by some compilers. Although such prohibition might make sense in terms of trying to achieve maximum portability, it is probably unrealistic to go this far.

2.5. Sufficient Precision for Fixed-Point Values

The Ada RM is silent on the maximum precision required for fixed-point values. Particularly in the case of fiscal calculations, this has the unfortunate consequence that Ada programmers cannot count on the availability of sufficient precision for fiscal calculations. Most implementations support only 32 bits of accuracy in fixed-point, which is clearly insufficient for fiscal purposes (COBOL requires a range of 10^{18}, corresponding to about 64 bits of precision).

Note that although fixed-point arithmetic is nothing more than scaled integer arithmetic, it is not easily possible to simulate k-bit fixed-point calculations using k-bit integer arithmetic, because in the case of multiplication of fractions, a double-length result is required. This means that the provision of 64-bit integer arithmetic is not sufficient to meet the requirement for fiscal calculations, even if a programmer were willing to do all the scaling manually.

POSSIBLE ADA 9X REQUIREMENT

Ada 9X should provide sufficient minimum fixed-point precision to accommodate the needs of fiscal calculations.

Discussion: Although Ada was originally intended to be restricted to the area of embedded applications, it is clear that the language has much wider applicability. There is already a significant activity in creating fiscal applications in Ada, both in commercial contexts, and in Department of Defense applications. Ada is in many ways ideally suited for such applications except in the area of fixed-point precision and decimal small implementation.

It is possible that this requirement could be considered to be an extra-language feature, provided for example by a set of usability guidelines. However, it is clearly more desirable from a uniformity point of view if it is a language requirement. Implementing 64-bit fixed-point is a relatively minor requirement for compilers. It typically means that a set of 128-bit arithmetic run-time routines have to be written, which is reasonably straightforward. Note by contrast that requiring 64-bit integer precision is much more contentious, since it implies that these 64-bit integers must be usable as array subscripts, loop indices etc.

Compatibility considerations: None. The suggestion is to mandate a minimum required precision. Since none is mandated currently, the result would be strictly upwards compatible.

Another issue with respect to precision arises with respect to high precision timers. On a machine whose timer resolution is considerably finer than 50 microseconds, 32 bits are not enough for the type DURATION.

POSSIBLE ADA 9X REQUIREMENT

Ada 9X should provide sufficient minimum fixed-point precision to accommodate the representation of DURATION values for high resolution timers.

Discussion: Probably 48 bits are sufficient for this purpose. It is clear that there are situations where 32 bits are insufficient.

Compatibility considerations: None. The suggestion is to mandate a minimum required precision. Since none is mandated currently, the result would be strictly upwards compatible.

2.6. Packed Decimal and Display Arithmetic Formats

COBOL requires the implementation of arithmetic which works on integers represented as strings of ASCII digits, called "Display" format in COBOL. The advantage of this format is that external files containing numbers in this format are entirely machine independent.

In addition, although not required by the COBOL standard, typical COBOL compilers implement an additional format for scaled integers called COMPUTATIONAL-3, which is packed decimal (one digit stored in four bits, two digits to a byte). This format is supported at the hardware level by many mainframe computers, including the IBM 370.

An issue arises of whether Ada implementations can or should support either or both of these formats.

Although there is no reason not to use packed decimal in an Ada implementation, there is a strong assumption that the underlying implementation of fixed-point will be in binary, and this is codified in 3.5.9(6), which defines the model numbers as covering a binary range, described by the mantissa value B. This means that if we declare a fixed-point type:

```
type COST is delta 0.01 range -999.99 .. +999.99;
for COST'SMALL use 0.01;
```

then it is natural to consider implementing COST using a three-byte packed decimal format (five nibbles contain the five digits, and one nibble contains the sign, using for example the standard IBM 370 format).

However, consider an expression such as:

```
A := B + C + D;
```

The intermediate value B + C cannot be stored in this same three-byte format because 3.5.9(6) requires that the mantissa be 17 bits, corresponding to a range for the base type of

$$-1310.78 \ldots +1310.78 \quad (131078 = 2^{17})$$

This means that the implementation is *not* justified in raising overflow for intermediate results which are outside the declared range but inside the extended range of the base type. Consequently an implementation would be forced to use a larger, e.g., 4-byte format, for the intermediate result, and then check the range only on the final assignment.

It should be noted that the requirement of using extra precision for the intermediate results is not necessarily an unacceptable one. The corresponding COBOL situation

```
ADD B C D GIVING A
```

where all four data names have the picture S999V99, actually requires that the intermediate values be held in sufficient precision to guarantee that no intermediate overflow can occur. In other words, COBOL has rather *more* strenuous requirements than Ada in this respect. Generally it is desirable for intermediate results *not* to overflow, so the Ada restriction is not necessarily a burdensome one.

Note also that we are not in a situation where intermediate results are held in registers with fixed precision, so the requirement is not as disturbing as it would be in the binary integer

case. It makes very little difference to the compiler, or to the efficiency of the executing program, whether the intermediate value is allocated as a four-byte packed decimal value or a three-byte packed decimal value. The Ada assignment:

```
A := B + C + D;
```

can generate four instructions on a typical machine supporting packed decimal, as follows:

```
Move B from 3-byte packed field to 4-byte temporary
Add packed C into the 4-byte temporary
Add packed D into the 4-byte temporary
Move the 4-byte temporary to the 3-byte result with overflow check
```

Note that we cannot in any case in Ada use A as the temporary location to compute the result, since exception semantics say that A should not be modified if the result is out of range.

It is important to note that there is no particular relation between the use of packed decimal and decimal *small* values. Packed decimal is simply another way of representing integers, and theoretically it is perfectly possible to use packed decimal for any fixed-point values, including those with binary *small* values. However, in practice, packed decimal is most relevant for fiscal applications where decimal *small* values are likely to be used.

The issue for Ada 9X is whether to consider any uniform syntax for the implementation of display arithmetic or packed decimal. Such a feature would most likely be optional, but there still might be some value in enforcing a uniform syntax for those compilers supporting this feature.

POSSIBLE ADA 9X REQUIREMENT

Ada 9X should support interoperability with COBOL data files, at least on an optional basis, including support for packed decimal and display arithmetic formats.

Discussion: Typical fiscal calculations involve access to existing data files. If these files have been created using COBOL programs, or other database applications using similar formats, they are likely to contain either display format or packed decimal data items. An Ada program will be able to interact with such files much more naturally if the Ada compilers support these formats.

Compatibility considerations: None. This is a proposed upwards compatible extension.

2.7. Ordering of Representation Clauses

The representation clause for *small* is anomalous in that it does much more than specify a representation; it also affects the set of values and the semantics of operations on the type.

One consequence of this is that problems arise if representation clauses for *small* appear too late. One example is:

```
type FIXED_6 is ....

-- define subprograms using FIXED_6
```

```
type FIXED_7 is new FIXED_6;
for FIXED_7'SMALL use ...
```

The current language definition permits this sequence,[3] but the consequence would be unexpected complex implicit conversions when the derived subprograms are called. This seems clearly undesirable. A length clause specifying *small* should not be allowed in this context.

Another example is:

```
type FIXED_8 is ...
subtype FIXED_9 is FIXED_8 range X .. Y;
for FIXED_8'SMALL use ...
```

The validation of the range X .. Y uses the safe numbers of the type, but these values are potentially affected by the later clause specifying FIXED_8'SMALL. Although the appearance of FIXED_8 in a subtype declaration is not a forcing occurrence, it is clear that the *small* declaration appears too late to be interpreted using a linear elaboration model.

A similar issue arises with SIZE clauses for implementations that left-justify fixed-point values. Consider the following:

```
type FIXED_9 is delta 0.25 range -1.0 .. +1.0;
for FIXED_9'SIZE use ...
```

As previously discussed, an implementation is currently free to choose SAFE_SMALL much finer than 0.25 for the first declaration. However, if this is done, then the SIZE clause has the unexpected property of changing the set of values by removing the extra precision.

POSSIBLE ADA 9X REQUIREMENT

The situation with respect to representation clauses for fixed-point must be clarified. In particular, the positioning of SIZE and SMALL clauses is problematic.

Discussion. In some respects, it would have been better if the SMALL clause had been part of the type declaration, or if the distinction between DELTA and small were removed from the language. However, such major changes are presumably out of bounds for Ada 9X. Clearly from the examples, some restrictions are necessary. One possibility is simply to forbid any occurrence of SMALL clauses after any mention at all of the type involved. Note that the issue of left versus right justification of fixed-point values (i.e., whether extra precision is provided where possible) interacts with this discussion, in that SIZE clauses are also problematical for implementations which left-justify fixed-point values (providing the extra precision).

Compatibility considerations: It is not clear now what implementations do when confronted with examples of the type given. The ACVC suite has been careful to avoid such cases, since the situation is unclear. Probably there are already portability problems between implementations in this regard, so it is likely that Ada 9X cannot succeed in being compatible with all existing implementations.

[3] RM 13.1(3) says: "A length clause is the only form of representation clause allowed for a type derived from a parent type that has (user-defined) derivable subprograms." Since a representation clause for *small* is a length clause, the above sequence of declarations is allowed.

3. Fixed-Point Computations

The following types of fixed-point calculations are provided in the current language. All calcuations are performed in terms of safe numbers, i.e., in terms of the model numbers of the fixed-point base type.

1. Fixed-point addition and subtraction of values of the same type. Since the inputs and outputs are all safe numbers, the results are precisely dictated. The operations correspond simply to integer addition and subtraction, so there are no implementation or semantic difficulties.
2. Multiplication or division of two fixed-point numbers, which can have different types, to give a resulting fixed-point number, which can have a third type. The result is required to be accurate in the safe number sense, i.e., if the result is a safe number, it must be accurate; otherwise it must lie in the corresponding interval. Since the safe numbers are separated by units of SAFE_SMALL, there are no machine numbers between safe numbers, so there are only two possible results, corresponding to rounding-up or rounding-down. Implementations are free to choose whichever they like, and there is not even a consistency requirement.
3. Multiplication or division of two fixed-point numbers, which can have different types, to give a resulting integer. In this case, the result must be accurately rounded.
4. Multiplication or division of two fixed-point numbers, which can have different types, to give a resulting floating-point value. In this case the result must be safe number accurate *from the point of view of the floating-point result type.*
5. Conversion of one fixed-point type to another. The result must be safe number accurate.
6. Conversion of fixed-point numbers to integer. The result is required to be accurately rounded.
7. Conversion of fixed-point numbers to floating-point. The result is required to be safe number accurate from the point of view of the floating-point result type.
8. Input-output of fixed-point numbers using TEXT_IO.

We treat these cases in turn, considering possible problems and shortcomings in the current approach.

3.1. Fixed-Point Addition and Subtraction

The only issue here is that the requirement that the two operands have the same type is inconvenient for programmers. COBOL programmers are used to simply writing:

```
ADD INPUT-1 INPUT-2 GIVING RESULT-1
```

where the scale (i.e., the SAFE_SMALL) values of the input and result data items can be different. An Ada programmer would be forced to write something like:

```
RESULT_1 := RESULT_1_TYPE (INPUT_1 + INPUT_1_TYPE (INPUT_2));
```

Furthermore, careful thought must be given to the exact choice of conversions. The apparently equivalent forms:

```
RESULT_1 := RESULT_1_TYPE (INPUT_2_TYPE (INPUT_1) + INPUT_2);
RESULT_1 := RESULT_1_TYPE (INPUT_1) + RESULT_1_TYPE (INPUT_2);
```

are not equivalent in terms of intermediate overflow. It is possible to construct three examples where the RESULT_1 value is in range of RESULT_1_TYPE, but only one of the above forms avoids intermediate overflow (a different one in each case). It is always the case that one of these forms works correctly, but figuring out which one depends on the particular data values involved, so there is no general programming approach to solve the problem. Furthermore, as soon as explicit conversions are involved, the conversions are only safe number accurate, and thus allow implementation-dependent rounding to be introduced into the calculation.

By contrast, a COBOL programmer can write either:

```
ADD INPUT-1 INPUT-2 GIVING RESULT-1
ADD INPUT-1 INPUT-2 GIVING RESULT-1 ROUNDED
```

with the assurance that if the result is in range, it will be correctly computed regardless of the scales involved, with accurate truncation in the first case, and accurate rounding in the second case.[4]

POSSIBLE ADA 9X REQUIREMENT

Ada 9X should consider extending the fixed-point addition and subtraction operators to allow mixing of types, as is permitted for multiplication and division.

Discussion: This would provide a level of convenience comparable to that available to COBOL programmers, and would be more consistent with the treatment of division and multiplication. Rounding and truncation issues would have to be addressed, as would the possible computational difficulties in the case of peculiar mixed smalls (this point is discussed in the context of multiplication and division).

Compatibility considerations: None. If an extension is provided here it would be upwards compatible, since it would allow notations which are currently illegal in Ada.

---

[4]In general, COBOL fixed-point addition and subtraction must be performed with more precision than that associated with the operand types. For example, INPUT-1 and INPUT-2 must be converted to higher precision before computing the sum, which must then be converted to the result type. Such conversions cannot always be specified in Ada, because the programmer cannot always declare types with the required precision. For example, if the maximum precision for a declared fixed-point base type is k, some calculations will need to be performed with $2k$ bits of precision, and by definition, such types cannot be declared.

3.2. Fixed-Point Multiplication and Division

There are three issues here, two minor and one rather complex.

3.2.1. Implicit Scaling Conversions

The first issue involves the following convenience. In Ada, one cannot write:

```
A := B * C;
```

where A, B, and C are all fixed-point values, even if they are the same type. Instead, the conversion must always be written:

```
A := FIXED_POINT_TYPE (B * C);
```

The requirement for the conversion in this case seems particularly annoying, and is puzzling to Ada programmers encountering fixed-point for the first time, since it seems so inconsistent with the treatment of integer and floating-point operations.

Similarly, one cannot write:

```
A := 2.0 * B;
```

or even

```
A := FIXED_POINT_TYPE (2.0 * B);
```

Instead, the required form is:

```
A := FIXED_POINT_TYPE (FIXED_POINT_TYPE (2.0) * B);
```

which seems even more ridiculous to the uninitiated. In these simple cases, it certainly seems clear that the explicit conversions are redundant. The trouble is that generalizing this observation runs into difficulties. If we allow:

```
A := B * C / D * E;
```

where A, B, C, D, and E are all different types with different SAFE_SMALL values, then it is quite unclear what the rules should be for intermediate precisions. It is instructive to look at what is done in other languages allowing fixed-point.

In COBOL, forms like:

```
MULTIPLY INPUT-1 BY INPUT-2 GIVING RESULT-1
```

allow free mixing of scales (though of course the scales are limited to decimal values), and the result is precise if it is in range, truncated or rounded as specified by the programmer. However, if a COBOL programmer writes:

```
COMPUTE A = B * C / D * E
```

then the COBOL standard has nothing at all to say about how this computation is carried out; it is simply said to be "implementation dependent." COBOL implementations use various approaches to handle intermediate scaling (one approach is to use floating-point for COMPUTE statements) and there are resulting portability problems.

COBOL programmers usually avoid the use of COMPUTE because of these uncertainties. It seems quite appropriate and natural for fixed-point programming to specify the intermediate precisions explicitly (unlike the case of integer and floating-point, where this is implicit).

In PL/1, general expressions are permitted, and there is a complicated set of rules that specifies exactly how intermediate scaling is carried out. However, these rules give rise to anomalies, such as the one which requires that the expression:

```
25 + 1 / 3
```

always overflow, and requires instead that this be written as:

```
25 + 01 / 3
```

In practice, these rules have not proved successful, and it is unlikely that any set of rules can be devised which avoids anomalies that are puzzling at best and incomprehensible at worst to programmers.

The Ada approach of allowing general expressions but requiring that the intermediate precision be spelled out is therefore quite reasonable except for the rather clumsy dictions required in the simple cases.

POSSIBLE ADA 9X REQUIREMENT

Ada 9X should examine the possibility of relaxing the requirements for explicit conversion in fixed-point multiplication and division.

Discussion: As described above, doing this in a general manner poses some nasty problems. On the other hand, solving only the "simple" cases threatens to introduce non-uniformities. The issue is whether there are partial solutions that allow nice dictions in the simple cases without introducing too much non-uniformity.

Compatibility considerations: None. This change would involve allowing some expressions and statements which are currently illegal in Ada.

3.2.2. Rounding and Truncation

The second issue involves rounding and truncation. The semantic model for fixed-point computations in Ada is deliberately similar to that for floating-point, and is phrased in terms of safe numbers and safe intervals. Consider the following example:

```
type FIXED_1 is delta 0.25 range -8.00 .. +7.75;
for FIXED_1'SMALL use 0.25;

A, B, C : FIXED_1;

B := 3.75;
C := 2.0;
A := FIXED_1 (B / C);
```

The true result, 1.875, is not a safe number, so the result is required to be in the safe interval containing 1.875. This interval has two values in it, 1.75 and 2.00. Implementations are free to choose either of these two results, and it is in fact the case that current available Ada implementations are inconsistent in their choice of results.

As we have mentioned before, COBOL programmers have full control over whether results are truncated or rounded, and in either case the result in a COBOL program is totally specified in the standard.

It is clearly desirable that implementations give the same results for all computations unless

there are appropriate hardware-related reasons why this should *not* be the case. In the case of floating-point, there *are* such reasons, since we want Ada programs to use the available hardware floating-point, and it is known and understood that such matters as rounding of results *do* differ from one machine to another.

However, in the case of fixed-point, the underlying implementation is typically in terms of integer operations, and these are consistent from one machine to another. There seems no good reason to permit implementors variations here. Both truncation and rounding are easy to implement.

Note: there is one machine that has provision for hardware fixed-point, namely the Transputer, which has a fractional multiply instruction that gives a rounded result. However, even on the Transputer it is easy to implement truncation if this is desired, since there are also integer multiplication operations of the conventional type.

POSSIBLE ADA 9X REQUIREMENT

The implementation freedom in choosing fixed-point results should be eliminated; the language should specify whether results of fixed-point operations should be rounded or truncated.

Discussion: As described above, there is no good reason for allowing variations between implementations here. Once again this means that floating-point and fixed-point are semantically less similar than they seem, but this is quite appropriate, and indeed many of the problems with Ada fixed-point stem from an inappropriate (in retrospect) attempt to merge fixed-point and floating-point semantics. If a programmer must be allowed both the rounding and truncation possibilities, as in COBOL, extra declarative mechanisms (e.g., a pragma) must be added. If only one mechanism is provided, rounding is more advantageous from a programming point of view, but is likely to be slightly less efficient in some cases.

Compatibility considerations: Technically none, since this is just a matter of removing allowed non-determinism. It is conceivable that some programs might be affected, but unlikely.

3.2.3. Mixed Bases for *small*

The third problem with fixed-point multiplication and division is that there are computational complications when the *small* values of the input arguments and results are not simply related. In the case where all three have the same base for the *smalls* (e.g., all powers of 2 or all powers of 10), there are no difficulties. This is why COBOL has no problems in allowing the general case.

However, if different *smalls* are mixed, then the computations are more complex. Jean-Pierre Rosen [2, 3] has described the treatment of the case where *smalls* are allowed to be any combination of powers of 2 and powers of 5, and Paul Hilfinger [1] has treated the more general case. The conclusion is that it is possible to deal with the general case without undue computational inefficiency (only double precision is required), but the required computations are somewhat complex. Furthermore, if rounding is permitted or specified, then the mixed cases become even more complex from a computational point of view.

The issue here is whether these mixed cases are sufficiently important to be mandated. At the moment, the RM is phrased to suggest that an implementation may reject the declarations of *smalls* under some circumstances, but that once it has accepted them, then it must allow all computations on these values, including the difficult mixed *small* cases in multiplications and divisions. This seems unfortunate, since in practice, the major requirements are for the non-mixed cases (all binary *smalls* or all decimal *smalls*), where no problems arise.

The difficulty of implementing these cases has historically been the sticking point for many implementations in allowing the declaration of *smalls* at all. We still have many implementations that do not permit the simple cases of decimal *smalls*, and this is likely to be a substantial part of the reason, since the implementation of the simple cases is straightforward (it is after all required in all COBOL compilers).

We defer stating a possible requirement in this area until we have considered the next two situations which are even more annoying.

3.2.4. Fixed-Point Multiplication and Division Yielding an Integer Result

The major issue here is similar to the third point discussed in the previous section except that the computational implications of the mixed *small* case are even worse here. The problem is that Ada *does* specify accurate rounding in this case. Making sure that the rounding is absolutely precise in the case where the *smalls* are unrelated (e.g., one operand has a *small* of 1/3 and the other of 1/7) is computationally complex. Paul Hilfinger has addressed this issue but his treatment is incomplete (in that it restricts the range of allowable *smalls*) and the resulting computation sequences are unpleasantly complex.

The motivation behind specifying rounding in the Ada definition is clear enough. It is simply a special case of the general principle that conversion of a fixed-point value to integer should be rounded. However, it is not at all clear that the definition in the RM really intends to introduce the level of complexity implied by requiring rounding in the multiplication and division cases.

3.2.5. Fixed-Point Multiplication and Division Yielding a Floating-Point Result

Again, the problematic cases arise in the case of mixed *small* bases. The result is required to be safe number accurate in terms of the floating-point type. In some implementations, the most accurate floating-point type can have many more bits of precision than the most accurate fixed-point type. In some cases, the required accuracy can be obtained by doing the operation entirely in floating-point, but it is by no means clear that this is always the case. Thus there is a possibility of this requirement implying the need for high precision integer arithmetic, just to handle this case.

Here again, the requirement for safe number accuracy is simply a special case of the general Ada principle of requiring safe number accuracy for floating-point operations, but it is not clear that the complexity in the fixed-point multiplication and division case is intended or

desirable. Jean-Pierre Rosen treats this case for his limited model where *small* values are restricted to a combination of powers of 2 and powers of 5. In this model, fixed-point values are 32 bits, and all operations are required to be computable with double-length arithmetic (64 bits). Under these restrictions, his model is only able to handle digits 6 and still guarantee safe number accuracy. Paul Hilfinger does not treat the floating-point case in the currently available version of his paper.

POSSIBLE ADA 9X REQUIREMENT

Ada 9X should clarify the cases of fixed-point multiplication and division that an implementation is required to handle and ensure that no unduly complex arithmetic requirement is implied.

Discussion: This requirement is related to the issue of what values of small an implementation is required to support. The current Ada definition creates a situation where the range of small values accepted by a compiler tends to be restricted by the difficulty of implementation of the mixed small cases for division and multiplication. Such mixed small cases are (a) not often required and (b) it is not clear that the accuracy requirements, particularly in the integer and floating-point result cases, are really required. One possible approach is to make it clear that certain "unpleasant" mixed small cases can be rejected at compile time, even if the representation clauses for the small values have been accepted. The rationale for Ada 9X should include a detailed description of the arithmetic procedures required for fixed-point operations. As far as possible, these operations should not require more than double-length arithmetic.

Compatibility considerations: The likely situation in Ada 9X is that a certain minimum set of capabilities is required of all compilers. The semantics of this prescribed set of capabilities should be identical to current Ada. Compilers can then provide additional facilities as they please.

3.3. Fixed-Point Conversions

The conversion of one fixed-point type to another again requires safe number accuracy. The required calculations are slightly tricky in some cases, but it does appear that all possible conversions can be carried out using only double-length arithmetic, so there is no great problem here. The Ada 9X rationale should include a description of the approach required here, and verify that this is in fact the case.

Conversion of fixed-point to integer requires accurate rounding. This may be problematic for some peculiar *small* values. The calculation required is a simple multiplication and division, but if the *small* is given as the ratio of two very long numbers, getting the rounding right may be tricky.

Conversion of fixed-point to floating-point requires safe number accuracy for the result. Again this may be tricky in some cases of peculiar *small* values.

3.4. Input-Output of Fixed-Point Values Using TEXT_IO

The issue here is whether the output has to be accurate in the case where a very large number of digits is specified. For instance, if the *small* value is 1/7, it seems to be required that if the value 1/7 is output with 100 digits after the decimal point, then these digits must be the accurate decimal expansion of 1/7. This requires unnecessary complexity in the conversion routines, and is probably not intentional. Note that a similar problem arises with the output of floating-point safe numbers to "excessive" accuracy.

POSSIBLE ADA 9X REQUIREMENT

The accuracy requirements for TEXT_IO output of fixed-point and floating-point values should not imply unnecessary complexity in the conversion routines.

Discussion: Since the required precision of output values is not known until execution time, there is a concern that all programs may have to pay time and space penalties for conversion routines that provide "excessive" accuracy. There is a test in the ACVC 1.11 suite that precisely tests this situation in the floating-point case, and it has been protested by at least one implementor. There is no major problem in providing conversion routines which provide the required unlimited accuracy, but they are likely to be big and slow, and it is not clear that it is sensibly useful to be able to output numbers accurately with many more digits than makes sense.

Compatibility considerations: Probably none. In theory, there could be programs which expect full accuracy, and if Ada 9X relaxes this requirement, such programs could malfunction. However, most current compilers do not in any case provide this functionality, and it is most unlikely that there are programs of this type around.

References

1. Hilfinger, P. Implementing Ada Fixed-Point Types Having Arbitrary Scales. University of California, Berkeley, March, 1990.

2. Rosen, J.-P. "Arithmétique réele en Ada". *Actes des journées Ada AFCET+ENST Bigre*, 42 (Dec. 1984).

3. Rosen, J.-P. *Une machine virtuelle pour Ada: le système d'exploitation.* Ph.D. Th., ENST, 1986. Paris, France.

Appendix A: Summary of Recommendations

If an explicit specification of *small* is given for a fixed-point type, it should determine the machine precision used to hold all values of the type. No intermediate results should be held with increased accuracy (see page 4).

Ada 9X should be specify whether extra bits in fixed-point base types are used to give increased accuracy or increased range (see page 7).

Ada 9X should clarify the status of reduced accuracy representations for fixed-point types. In particular, Ada 9X should consider whether to require that SAFE_SMALL for a fixed-point type determines the machine precision used for all values of the type (both stored and computed) (see page 8).

The range of required *small* values should be clearly specified. Ada 9X should support the requirements of fiscal calculations with regard to the implementation of decimal *small* values (see page 9).

Ada 9X should provide sufficient minimum fixed-point precision to accommodate the needs of fiscal calculations (see page 10).

Ada 9X should provide sufficient minimum fixed-point precision to accommodate the representation of DURATION values for high resolution timers (see page 10).

Ada 9X should support interoperability with COBOL data files, at least on an optional basis, including support for packed decimal and display arithmetic formats (see page 12).

The situation with respect to representation clauses for fixed-point must be clarified. In particular, the positioning of SIZE and SMALL clauses is problematic (see page 13).

Ada 9X should consider extending the fixed-point addition and subtraction operators to allow mixing of types, as is permitted for multiplication and division (see page 16).

Ada 9X should examine the possibility of relaxing the requirements for explicit conversion in fixed-point multiplication and division (see page 18).

The implementation freedom in choosing whether fixed-point results are rounded or truncated should be eliminated (see page 19).

Ada 9X should clarify the cases of fixed-point multiplication and division that an implementation is required to handle and ensure that no unduly complex arithmetic requirement is implied (see page 21).

The accuracy requirements for TEXT_IO output of fixed-point and floating-point values should not imply unnecessary complexity in the conversion routines (see page 22).

Appendix B: Implementation of Fixed-Point Arithmetic in the Low-Level Ada/ED Interpreter

J-P. Rosen
ADALOG
Paris, France

This Appendix is mainly a translation of a chapter of Rosen's thesis [3], with some adaptation and the removal of some points that were too specific to Ada/ED. The translation was prepared by Jean-Pierre Rosen and edited by John Goodenough.

Note: In this Appendix, all explanations are given with the assumption that all operands are positive except when the sign is treated independently and operations are performed on absolute values. The implementation was for a DEC/Vax with a 32-bit floating-point type. In Ada terms, this type has a mantissa of 21 bits (since the base is 16), meaning the corresponding DIGITS attribute has the value 6.

B.1. The Representation of Fixed-Point Numbers

Fixed-point numbers are defined as having the form *sign* * *mantissa* * *small* [RM 3.5.9(4)]. Since the value of *small* is a property of the fixed-point type, it is not necessary to keep it in every value; keeping it in the type template[5] is sufficient. A variable of a fixed-point type will hold only the signed mantissa, which is a plain integer value. This integer value is called the *representation* of the fixed-point value.

B.1.1. Representation of SMALL

The language authorizes a representation clause for SMALL, each implementation being free to choose what values are allowed for such a clause. The initial requirement for the low-level Ada/ED interpreter was to allow values of SMALL that were powers of 2 or powers of 10. Powers of 2 are mandatory (the implicit value of SMALL in the absence of a representation clause is always a power of 2 [RM 3.5.9(5)]), and it is likely that powers of 10 will be widely used, especially for programs that want to use fixed-point values to represent "dollars and cents," for example.

[5] A *type template* is an execution-time descriptor in the Ada/ED system.

The first implementation approach in Ada/ED used two positions in the type template, one to keep the base (2 or 10), and the other to store the power of the base. However, manipulating two kinds of fixed-point types led to a number of problems for conversions, mainly for multiplication and division of two values with different bases for SMALL, namely, one with a SMALL of 2^n and the other with a SMALL of 10^n. In this case, the result belonged to a type whose SMALL was neither a power of 2 nor a power of 10. This resulted in an extra conversion, possibly leading to a loss of accuracy. In a sense, multiplication and division were external operations over the set of fixed-point values defined this way. The solution was to make a closure of the set of fixed-point values by allowing for SMALL any value of the form $2^p \cdot 5^q$, therefore including powers of 2 and powers of 10 as special cases. This solution had the following benefits:

- Greater simplicity and efficiency in handling fixed-point operations;
- Unification of fixed-point types;
- Wider set of allowed values for representation clauses of SMALL;
- Easier formatting for printing.

To give an idea of the improvement provided by our solution, the following table lists the set of allowed values for SMALL with up to 3 fractional digits:

0.001 .. 0.010	0.010 .. 0.100	0.100 .. 1.000
0.001 *	0.01 *	0.1 *
0.002 *	0.016	**0.125**
0.004	0.02 *	0.128
0.005 *	0.025 *	0.16
0.008	0.032	0.2 *
	0.04	**0.25** *
	0.05 *	0.256
	0.064	0.32
	0.08	0.4
		0.5 *
		0.512
Bold: 2^x		0.625
Underlined: 10^x		0.64
*: *Simple fractions of* 10^x		0.8

Figure B-1: Allowed Values of SMALL of the form *0.ddd*

There are 28 allowed values, among which only 3 are powers of 2 and 3 are powers of 10. 12 of them are simple fractions of a power of 10, and therefore considered especially useful in practice.

Note that an implementation that would allow for arbitrary values of SMALL must keep them at execution time. But since SMALL is of arbitrary precision, these values must be kept as rational numbers at execution time, which can be expensive. Keeping them as floating-point values would mean restricting the allowed values of SMALL to the model numbers of the floating-point type. In contrast, the Ada/ED implementation makes it possible to reconstruct the rational form if necessary, while needing only two bytes in the type template (values for p and q), as in the initial solution.

B.1.2. Representation of the Mantissa

The mantissa is actually a plain integer. It was decided to support 64-bit mantissa values in order to allow for a sufficient range even for small values of SMALL. This implied being able to handle 128-bit arithmetic for temporary results (multiplication and division). Extended arithmetic on integers is a standard problem and will not be considered further. We will just assume that routines for integer arithmetic on 64-bit values are available.

B.2. Conversions involving fixed-point values

There are three kinds of conversions involving fixed-point types: to or from other fixed-point types, to or from integer types, and to or from floating-point types.

B.2.1. Conversions to or from Other Fixed-Point Types

The only important parameter when dealing with conversions between fixed-point types is the SMALL of each type. Consider a conversion from a *source* type with SMALL S_s to a *target* type with SMALL S_t. Recall that the way SMALL is kept in the type template (as exponents of 2 and 5) is only a simplified way for keeping them as rational values. Since the model numbers of a fixed-point type are integer multiples of SMALL, converting a value, V, from a source type into a target type means finding a representation R_t (RM [4.5.7]) such that, for an initial source representation R_s:

$$R_t S_t \le R_s S_s < (R_t + 1) S_t$$

Each SMALL is fully characterized by p and q, the exponents of 2 and 5. To simplify later demonstrations, let p^+ be the value of p if it is positive, 0 otherwise; let p^- be the value of p if it is negative, 0 otherwise; and similarly for q. The conversion factor is a rational number, defined as:

$$\frac{N}{D} = \frac{S_s}{S_t} = \frac{2^{p_s^+}\ 2^{p_t^-}\ 5^{q_s^+}\ 5^{q_t^-}}{2^{p_s^-}\ 2^{p_t^+}\ 5^{q_s^-}\ 5^{q_t^+}}$$

The conversion is obtained by applying:

$$R_t = R_s \times N \div D$$

There is no rounding error as long as the multiplication is performed before the division.

As mentioned before, one of the requirements of the project was that all arithmetic be performed with 64-bit values; only intermediate results used 128 bits. There is no special problem with fixed-point arithmetic as long as SMALLs are powers of 2. Introducing arbitrary SMALLs induces converting factors that seem to require arbitrary length arithmetic. At the time of design, no compiler offered representation clauses for SMALL other than with powers of 2, and no literature could be found on this topic.

Since then, Froggatt [1] has demonstrated that triple-length arithmetic is sufficient to implement multiplication, division, and conversion between fixed-point values with arbitrary SMALLs. However, Froggatt couldn't apply his method (through the use of continued fractions) to conversions involving floating-point numbers, and the question of the size of the

required arithmetic to implement arbitrary fixed-point values is still open.[6]

Our goal was not to provide arbitrary SMALLs, but only the most useful ones, with the limitation of 128 bits maximum arithmetic. We therefore had to limit the allowed values for p and q. Since N and D in the above formula were explicitly reconstructed, they had to fit in 64 bits, which in turn implied that the numerator and the denominator of the rational representation of each SMALL had to be limited to 32 bits. Considering the initial requirement to support powers of 2 and powers of 10, this allowed SMALLs up to $2^{\pm 31}$ and $10^{\pm 9}$, which seemed sensible, considering the goals of the Ada/ED system. This implied limiting p to ± 31 and q to ± 9, with the supplementary constraint that if p and q are of the same sign, then $2^p \times 5^q$ must itself fit in 32 bits. Since 5^9 fits in 21 bits, a wider range could have been allowed for q, if not used for powers of 10 (i.e., with a smaller p). This did not seem useful at first, and was rejected on the ground that the justification would appear awkward to the average user. However, as will appear later, limiting q to ± 9 is mandatory to keep a 64-bit arithmetic.

B.2.2. Conversions to or from Integer Types

Fixed-point types have a natural relation to integer types. To unify the interface with fixed-point arithmetic routines, the system defines a type, called INTEGER_FIXED, defined as:

```
Min : constant := 1.0 * INTEGER'POS(INTEGER'FIRST);
Max : constant := 1.0 * INTEGER'POS(INTEGER'LAST);
type INTEGER_FIXED is delta 1.0 range Min .. Max;
```

INTEGER_FIXED is isomorphic to INTEGER (values of both types share the same representation); therefore, conversions between fixed-point and integer types are actually generated as operations with the type INTEGER_FIXED. There is, however, a subtle point here: conversions to INTEGER must round, while this is not necessary (but still allowed) for conversions to real types. This raised no problem in the Ada/ED system, since conversions are always rounded.

Conversions to or from integer types are therefore generated like any other fixed-point conversions, with the template for INTEGER_FIXED being used as the type template for integer values.

B.2.3. Conversions to or from Floating-Point Types

Conversion of a fixed-point value to a floating-point value uses the same algorithm as fixed-point division with a floating-point result (see B.3.3, below), since the conversion can be viewed as a division by the fixed-point value 1.0 with a floating-point result. The algorithm therefore rebuilds the rational form of the fixed-point number and then performs a *floating-point* division of the numerator by the denominator. More on this, especially accuracy considerations, will be found in section B.3.3.

Conversion from floating-point to fixed-point requires special care to guarantee the required

[6]This was written before P. Hilfinger's papers [2].

accuracy, especially when the resulting fixed-point value is equal to or near a model number of the floating-point type. First, note that a value of a floating-point type is defined (RM 3.5.7) as $sign \times mantissa \times 2^{exponent}$. This can be interpreted as a fixed-point value whose SMALL would be $2^{exponent-21}$, since *mantissa* is a number in the range 0.5 .. 1.0, with a representation in 21 bits (on our target machine, the Vax). Floating-point values that have a constant relative accuracy locally (i.e., that have a constant exponent) behave like fixed-point values. It is, however, not possible to apply the algorithm for fixed-point values, since the exponent may vary in a range of ± 84, which is much wider than our allowed range of ± 31 needed to guarantee that no overflow will occur during intermediate calculations. Let $M \cdot 2^e$ be the floating-point value to be converted. Let:

$$M' = M \cdot 2^{21}$$

If p and q are exponents of 2 and 5 for the (fixed-point) type of the result, converting the value means finding a representation R such that:

$$R = M' \cdot 2^{e-21} \frac{2^{p^-} \cdot 5^{q^-}}{2^{p^+} \cdot 5^{q^+}}$$

Let:

$$x = e - 21 - p$$

$$R = M' \frac{5^{p^-}}{5^{q^+}} 2^x$$

Since e belongs to the range ± 84 and p to the range ± 31, x belongs to the range -136 . +94. If q is positive and x negative, the numerator requires only the 21 bits for M'. If q is positive and x positive, the numerator fits in 21 + 94 = 115 bits. If q and x are both negative, the numerator fits in 21 + 31 = 52 bits. The numerator, for which 128 bits are allowed, can overflow only in the case where q is negative and x positive (21 + 31 + 94 = 146 bits). But in this case, no division is involved for the conversion. It is therefore possible to raise NUMERIC_ERROR (or CONSTRAINT_ERROR) safely as soon as the calculation overflows 128 bits. If x is positive, no accuracy is lost by computing $M' \times 5^{q^-} \div 5^{q^+}$, and then dividing by 2^x. This latter term may not fit in 64 bits, but the division can be performed by repeated shifts, and in effect, there is no need to compute 2^x.

B.3. Operations on Fixed-Point Numbers

Operations on fixed-point numbers include addition, subtraction, multiplication and division.

B.3.1. Additive Operators

Addition and subtraction are allowed only between operands of the same fixed-point type. They are treated as integer operations between mantissas, and involve no particular problem.

B.3.2. Multiplying Operators with Fixed-Point or Integer Result

Multiplication between a fixed-point value and a value of type INTEGER, as well as division

of a fixed-point value by a value of type INTEGER are allowed, and yield a value of the same type as the fixed-point value, without requiring any conversion. Note that such operations are allowed only with the type INTEGER, and not with any integer type. There is no conversion or accuracy problem with these operations, since they are simply performed on the representations as regular integer arithmetic.

Apart from this special case, multiplication and division are allowed between any fixed-point types. They yield a result of type *universal_fixed*. This type must in turn be *immediately* converted to some numeric type. This means that the result type is always known to the compiler, but it need not be a fixed-point type.

The case where the result is of a floating-point type will be dealt with separately later. If the result type is an integer type, it is performed on the type INTEGER_FIXED (see page 30), whose values have representations identical to INTEGER, with a DELTA (and a SMALL) of 1.0.

Multiplication of two fixed-point values yields (mathematically) a result whose SMALL is equal to the product of the SMALLs of both operands. Multiplication is performed the same way in the implementation: A temporary template is first built for the result, with a type size of 128 bits (since each operand may be 64 bits long) and exponents of 2 and 5 that are the sum of those of each operand; mantissas are then multiplied, and the result is converted into the required type, using normal conversion routines and the temporary template.

Division is rather more complicated, since the quotient of mantissas is not generally an integer, and the exact value of the quotient is required to guarantee the result's accuracy. We are now going to demonstrate that there exists a fixed-point type to which the dividend can be converted *before* doing the division, and that this type will yield enough accuracy for the result. This type is the one whose SMALL is equal to the product of the divisor's SMALL and the result's SMALL (the result's type is always known, [RM 4.5.5(11)]).

Let S_1 be the SMALL of operand 1, S_2 the SMALL of operand 2, and S_r the SMALL of the result. Similarly, let R_1, R_2 and R_r be the corresponding representations. By definition, R_r is such that:

$$R_r S_r \leq \frac{R_1 S_1}{R_2 S_2} < (R_r+1) S_r$$

Since the SMALL of the intermediate type, S_i, is equal to $S_2 \times S_r$, so converting R_1 into the intermediate type yields a representation R_i such that:

$$R_i S_2 S_r = R_1 S_1 \qquad (1)$$

As R_i is the result of an integer division:

$$R_i \leq \frac{R_1 S_1}{S_2 S_r} < R_i+1$$

Dividing by R_2 gives:

$$\frac{R_i}{R_2} \leq \frac{R_1 S_1}{R_2 S_2 S_r} < \frac{R_i+1}{R_2} \tag{2}$$

Let q be the quotient of R_i by R_2:

$$R_i = q R_2 + r$$

Equation 2 becomes:

$$q + \frac{r}{R_2} \leq \frac{R_1 S_1}{R_2 S_2 S_r} < q + \frac{r+1}{R_2}$$

Since $q \leq q + \frac{r}{R_2}$ and $q + \frac{r+1}{R_2} \leq q + 1$, multiplying by S_r gives:

$$q S_r \leq \frac{R_1 S_1}{R_2 S_2} < (q+1) S_r$$

Therefore, q, the result of the integer division of the intermediate representation by the representation of the divisor, is an acceptable value for R_r.[7]

Division operates between a 128-bit value and a 64-bit value, yielding 64 bits. Therefore, a template is first built with powers of 2 and 5 that are the sum of those of the second operand and of the result; the first operand is then converted into that type, then divided by the second operand. This ensures that the result automatically has the required type and accuracy. A check is then made that no overflow occurred; otherwise NUMERIC_ERROR (or CONSTRAINT_ERROR) is raised.

B.3.3. Multiplying Operators with Floating-Point Results

RM 4.5.5(11) says that the result of a product or quotient of fixed-point values must be converted to some numeric type, but there is no restriction on this type; it can be a floating-point type, in which case the previous algorithm cannot be applied.

In the case of a multiplication or division with a floating-point result, we rebuild the rational values of the operands and then multiply the numerators and the denominators (exchanging numerator and denominator of the second operand in the case of the division); we then divide the result's numerator by its denominator *in floating-point*.

Since the mantissa of a (normalized) floating-point number lies between 1/16 and 1 but the mantissa of a fixed-point number is an integer value, the floating-point mantissa must be shifted 21 positions to the left (on the Vax). Therefore, the mantissa of the result M is such that:

$$M = \frac{R_1 S_1 2^{21}}{R_2 S_2}$$

However, powers of 2 are part of the exponent part; the part that is not a power of 16 can be

[7]Thanks to G. Fisher for helping with this demonstration.

dealt with separately using shifts *after* the floating-point division. Therefore, the only computation that requires accuracy considerations is:

$$M = \frac{R_1 5^{q_1^+} 5^{q_2^-} 2^{21}}{R_2 5^{q_1^-} 5^{q_2^+}}$$

Since R_1 fits in 63 bits, and 5^q in 21 bits (with q limited to ± 9), the numerator requires $63 + 21 \cdot 2 + 21 = 126$ bits!

This is extremely close to the limits of our arithmetic, and actually works only because we limit floating-point accuracy to **digits** 6. Considering our requirement to stay within 128-bit arithmetic, allowing one more digit for floating-point would not be possible because of an overflow in *fixed-point* arithmetic.

An important conclusion (finding?) is that no proposed algorithm for fixed-point arithmetic can ignore its relation to floating-point since floating-point accuracy requirements are more demanding in the Ada model.

B.4. Fixed-Point I/O

The problem of I/O is rarely addressed in papers on fixed-point arithmetic. It is however an important problem since Ada requires the implementation of fixed-point I/O, and conversion to a printable form must not require excessive space or time-consuming algorithms.

The problem is not actually one of printing, but rather one of *converting a fixed-point value to a string representation.* The simplest algorithm would be to convert the value to floating-point and then use the regular translation routines for floating-point types. Unfortunately, the accuracy would be insufficient, especially for values located far from zero.

B.4.1. A Property of Our Fixed-Point Types

A consequence of our choice of supporting only numbers of the form $2^p \cdot 5^q$ for SMALL is that the number of significant digits that are needed to represent *exactly* any fixed-point value is finite. Moreover, the number of digits after the decimal point is a property of the *type* and can be determined at compilation time.

B.4.2. Demonstration

Let V be the value to be printed, R its representation, and p and q the exponents of 2 and 5 for its type.

$$V = R \cdot 2^p \cdot 5^q$$

If p and q are both positive, V is an integer. If only p is negative, then:

$$V = R \cdot 2^p \cdot 5^q$$

$$= R \cdot 5^{-p} \cdot (5 \cdot 2)^p \cdot 5^q$$

$$= R \cdot 5^{q-p} \cdot 10^p$$

As $R \cdot 5^{q-p}$ is an integer, this demonstrates that V has exactly $-p$ digits after the decimal point. Similarly, we can demonstrate that if only q is negative, V has exactly $-q$ digits after the decimal point.

If p and q are both negative, let us assert that $p > q$, i.e., $|p| < |q|$. The demonstration would be the same for the case where $p \leq q$.

$$V = R \cdot 2^p \cdot 5^q$$

$$= R \cdot 2^p \cdot 2^{-q} \cdot (5 \cdot 2)^q$$

$$= R \cdot 2^{p-q} \cdot 10^q$$

Since p is greater than q, $R \cdot 2^{p-q}$ is an integer, and the value has exactly q digits after the decimal point.

We have therefore demonstrated that every value of the form $R \cdot 2^p \cdot 5^q$ has exactly $\max(|p^-|, |q^-|)$ digits after the decimal point. This size depends only on the fixed-point *type*, not on the individual values, and can be determined at compilation type (p and q are static properties of the type).

B.4.3. Converting Fixed-Point Values

Let $r = \max(|p^-|, |q^-|)$. To correctly format a fixed-point value, simply multiply R by $10^r \cdot 2^p \cdot 5^q$ (yielding always an integer result), then format it using the regular integer formatting routine, placing a decimal point before the last r digits. If the value the user specified for AFT is less than the minimum required for the fixed-point type and rounding is necessary, just add $5 \cdot 10^{r-1}$ to $R \cdot 10^r$, and then set the r last digits of the result to 0.

Note that our method of formatting benefits from the fact that, unlike integer values [RM 14.3.7(2)], real values are always represented in decimal form [RM 14.3.8(2)], and that the "special" base used (10) is precisely the product of a power of 2 by a power of 5. Supporting other values for SMALL would require more investigation to properly format fixed-point values. To our knowledge, no literature addressing the problem is available.

B.5. Conclusion

Some of the design choices in this implementation could be questioned, the more restrictive one being limiting floating-point types to **digits** 6. We have seen that allowing a wider range would require either more than 64-bit arithmetic, or limiting more severely the range allowed for SMALL. Since Ada/ED is for educational purposes, greater accuracy in floating-point calculations was not felt to be of primary importance; giving extended facilities for fixed-point values was considered more important.

The goal of this Appendix was only to show that supporting SMALLs of the form $2^p \cdot 5^q$ provides a significant simplification of the algorithms and allows an implementation with a fixed, *a priori*, size for the arithmetic, while still providing the most useful values from the user's point of view.

It is certainly possible to implement full support for arbitrary values of SMALL; however, the extra effort required to support them should be balanced with considerations of the practical usefulness of such values.

References

1. Froggatt, T. Fixed-Point Conversion, Multiplication, & Division. System Designers PLC, April, 1986. Great Britain.

2. Hilfinger, P. Implementing Ada Fixed-Point Types Having Arbitrary Scales. University of California, Berkeley, March, 1990.

3. Rosen, J.-P. *Une machine virtuelle pour Ada: le système d'exploitation.* Ph.D. Th., ENST, 1986. Paris, France.